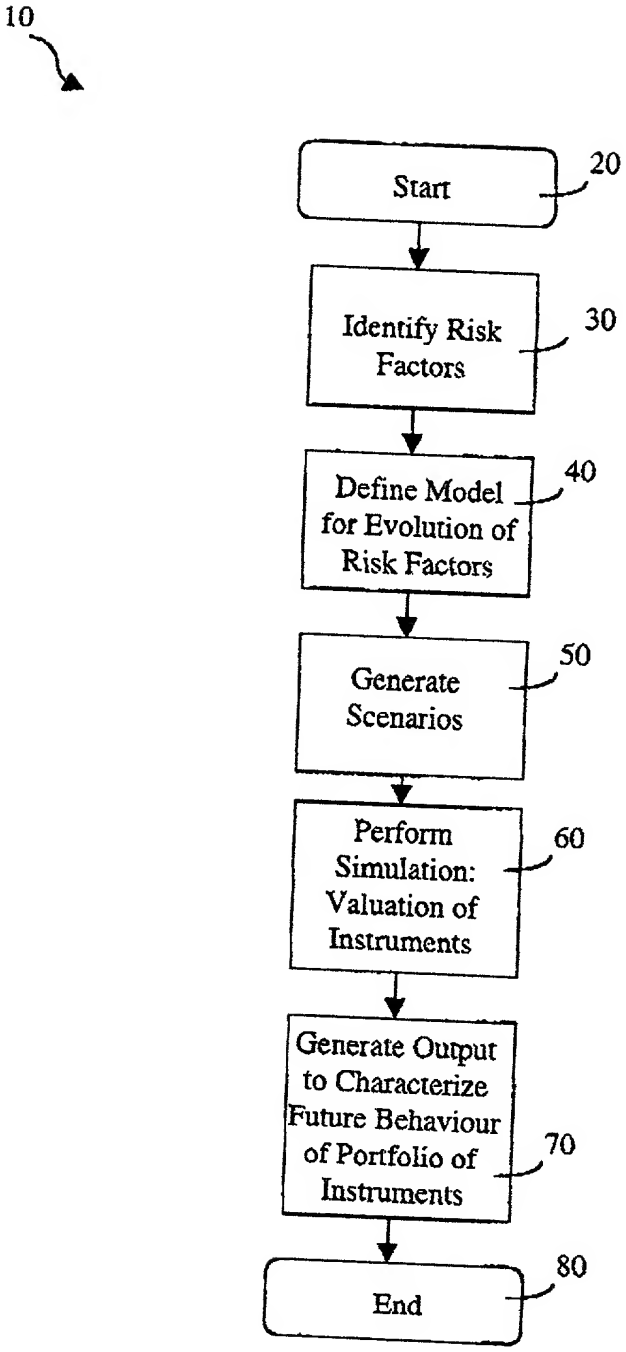
10
Start
20
Identify Risk Factors
30
Define Model for Evolution of Risk Factors
40
Generate Scenarios
50
Perform Simulation: Valuation of Instruments
60
Generate Output to Characterize Future Behaviour of Portfolio of Instruments
70
End
80

FIG. 1

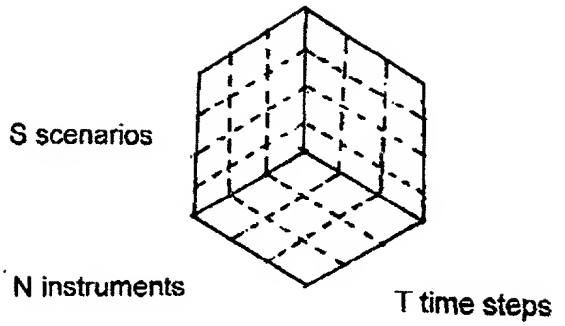

FIG. 2A

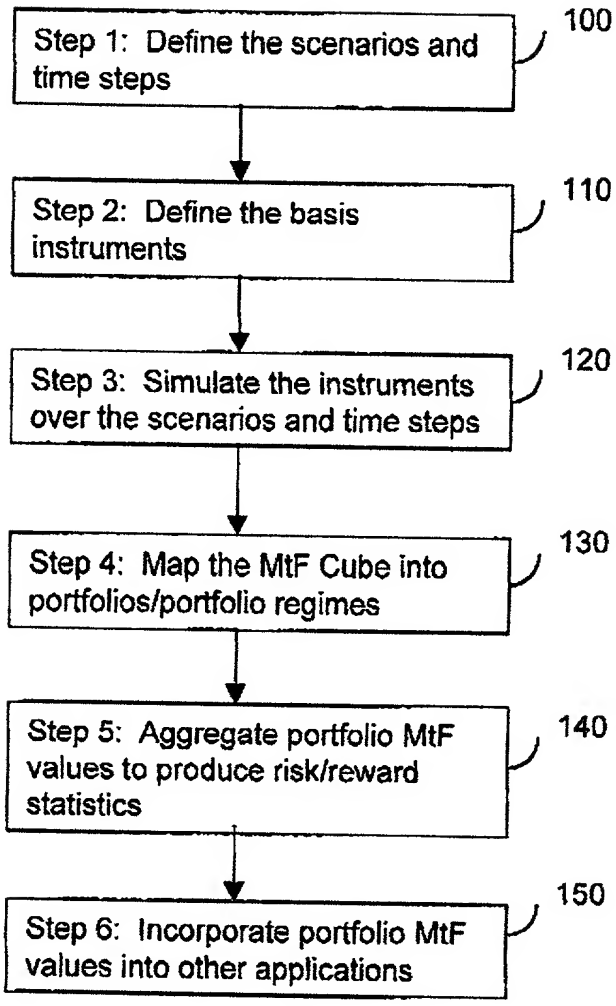
Step 1: Define the scenarios and time steps
100
Step 2: Define the basis instruments
110
Step 3: Simulate the instruments over the scenarios and time steps
120
Step 4: Map the MtF Cube into portfolios/portfolio regimes
130
Step 5: Aggregate portfolio MtF values to produce risk/reward statistics
140
Step 6: Incorporate portfolio MtF values into other applications
150

FIG. 2B

160

```
for (int i=0; i<n; ++i) {
    if (gen) {                          // Generator is not exhausted
        cout << "Element # " << i << "; "
            << "Generated value: " << *gen << ; "
            << "Weight: " << gen.getWeight() << ; "
            << "Accumulated Weight: " << gen.getAccumWeight() << endl
        ;
        ++gen;
    }
    else {
        cout << "The generator is exhausted. You may try to reset it."
            << endl
        ;
        break;
    }
}
```

FIG. 3

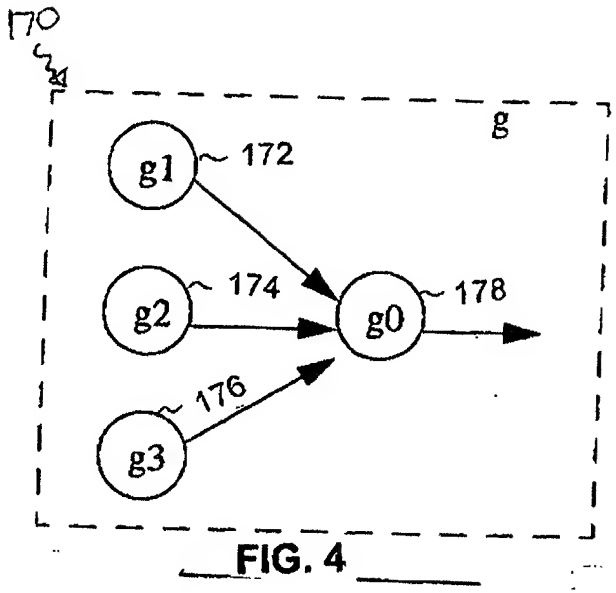

FIG. 4

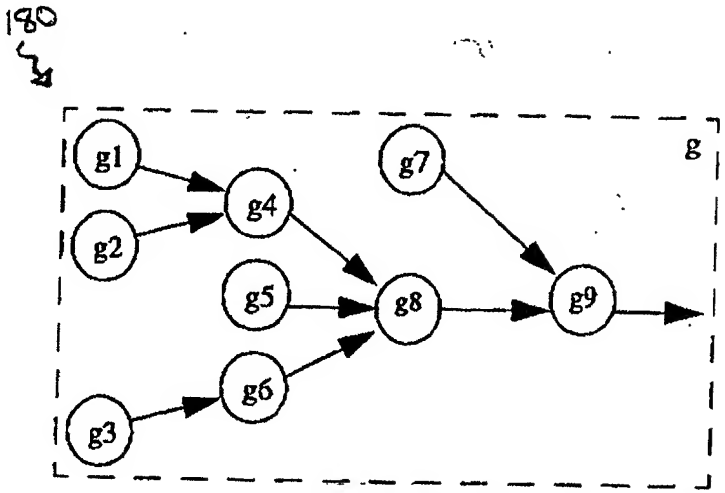

FIG. 5

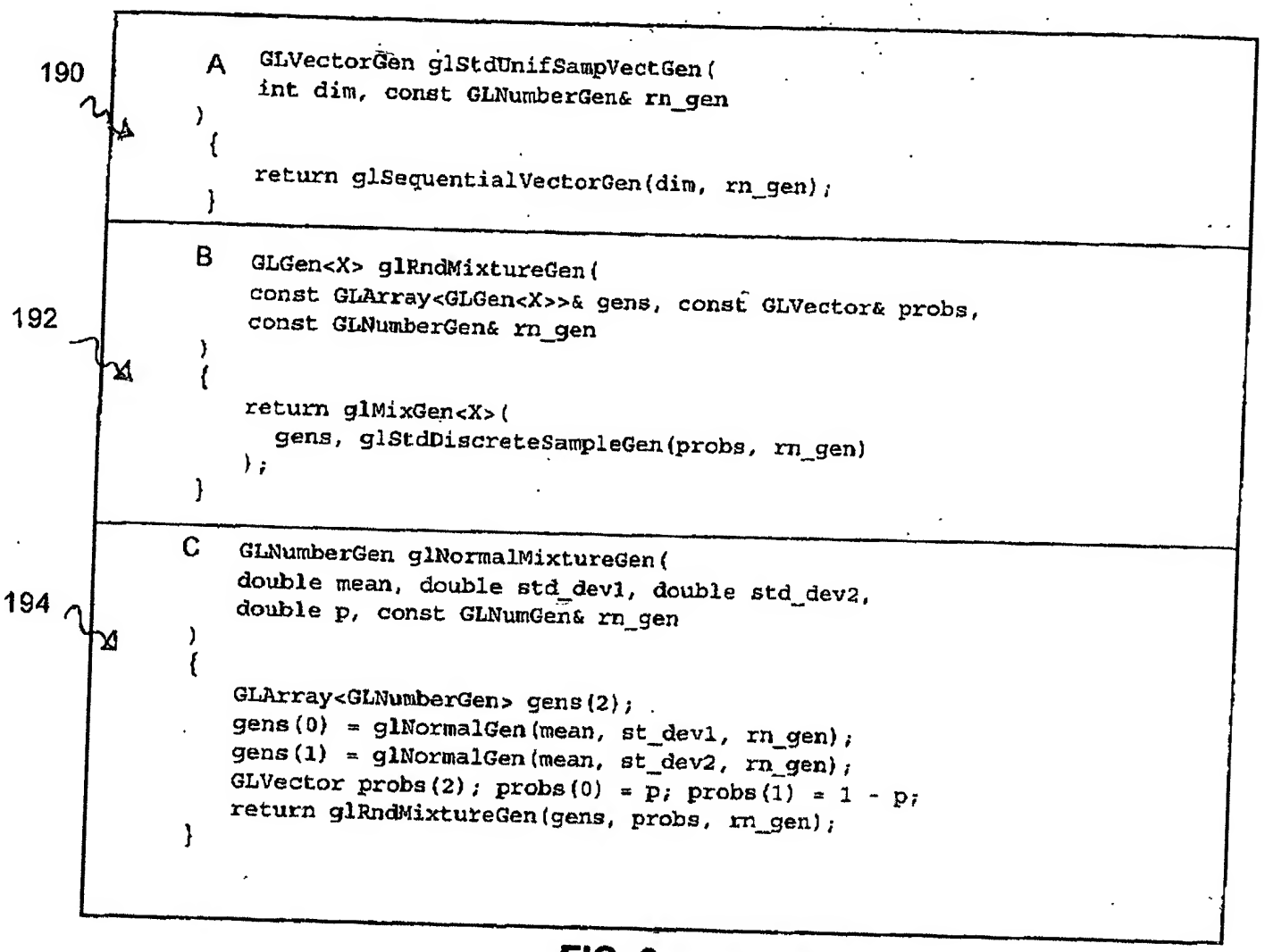

FIG. 6

200

```
GLVector x;
for (int i=0; i<100; ++i) {
    cout << "Input a vector: ";
    cin >> x;
    cout << "The image of the vector is " << m(x) << endl;
}
```

FIG. 7

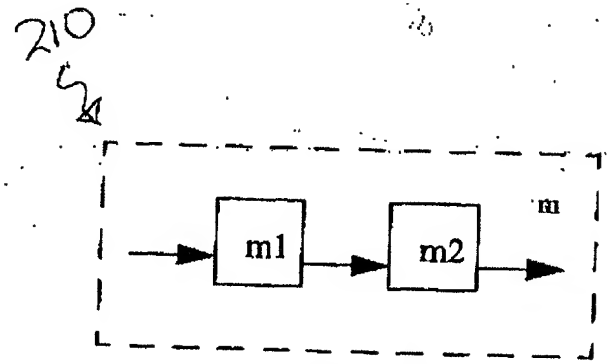

FIG. 8

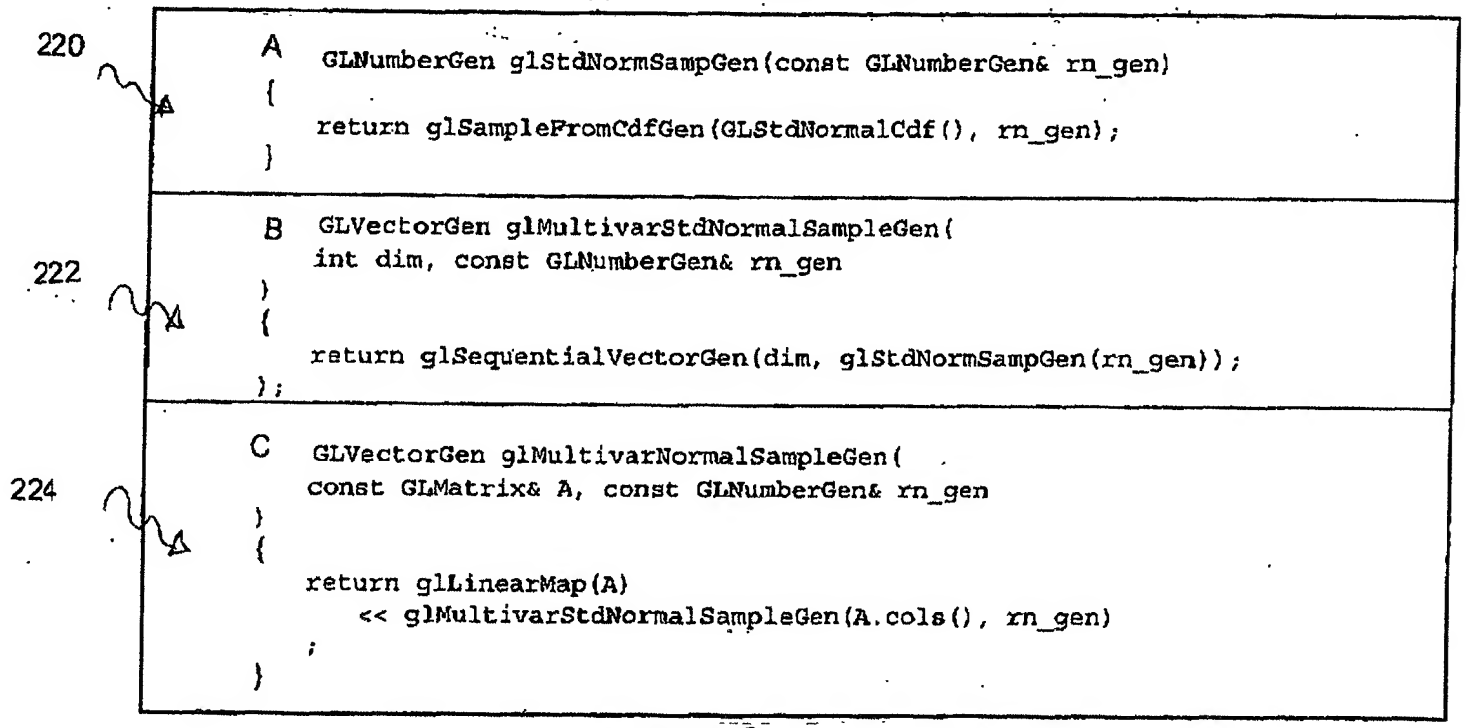

FIG. 9

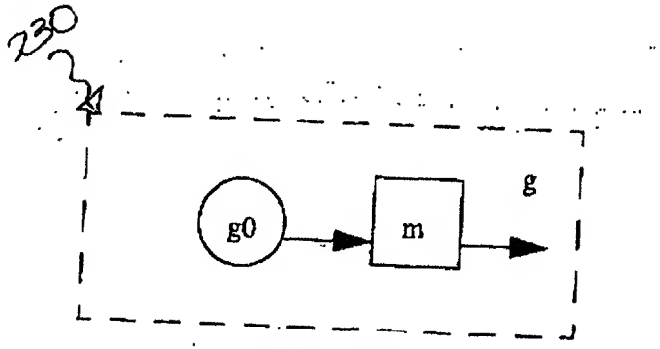

FIG. 10

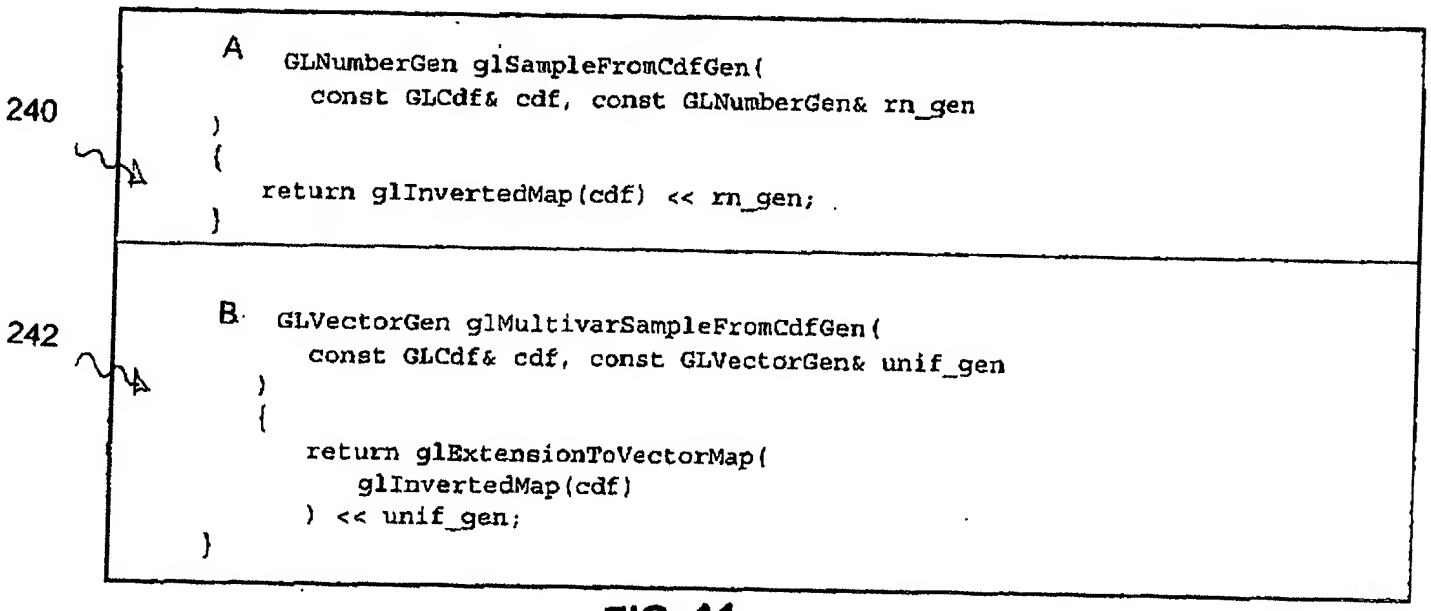

FIG. 11

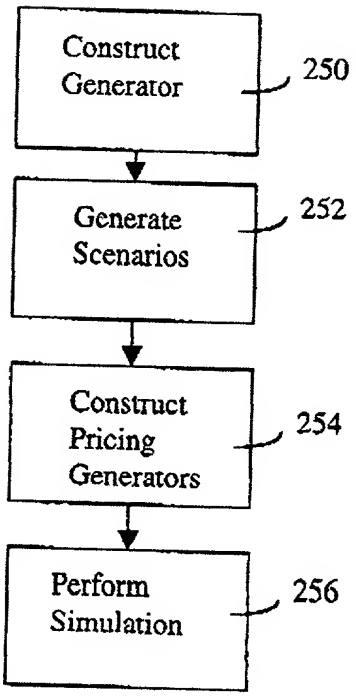
Construct Generator
250
Generate Scenarios
252
Construct Pricing Generators
254
Perform Simulation
256

FIG. 12

260

```
GLVectorGen getGeneralizedNormalGen(Interface& iface)
{
    switch (iface.getGenerationSchema()) {
    case PSEUDO_RANDOM_SAMPLING:
        int dim = iface.getDim();
        long seed = iface.getSeed();
        return glMultivarStdNormalSampleGen(dim, glRnGen(seed));
    case LOW_DISCREPANCY_SEQUENCE:
        int dim = iface.getDim();
        return glVectSampleFromCdfGen(
            glStdNormalCdf(0,1), glSobolSequenceGen(dim)
        );
    case STRATIFIED_SAMPLING:
        GLIntVector num_nodes = iface.getJamshidianNumNodes();
        return glJamshidianMultivarDistribGen(num_nodes);
    }
}
```

FIG. 13

270

```
GLVectorGen getLogNormalScenarioGen(Interface& iface)
{
    GLVector x0 = iface.getInitValue();
    GLMatrix A = iface.getTransformationMatrix();
    double dt = iface.getTimeStep();
    return x0 * glExtensionToVectorMap(glFromFuncPointerMap(exp))
        << glLinearMap(A*sqrt(dt))
        << getGeneralizedNormalGen(iface)
    ;
}
```

FIG. 14

280

A
```
GLNumberGen getInstrumentSimulationGen(
const GLGen<GLVector>& sc_gen, const GLFunc& pr_map
)
{
return pr_map << sc_gen;
}
```

282

B
```
GLVectorGen getByInstrumentPortfolioSimulationGen(
const GLVectorGen& sc_gen,
const GLArray<GLFunc>& pr_map_arr
)
{
return glScalarMergeMap<GLVector, double>(pr_map_arr)
    << sc_gen;
}
```

284

C
```
GLNumberGen getPortfolioSimulationGen(
const GLVectorGen& sc_gen,
const GLArray<GLFunc>& pr_map_arr, const GLVector& positions
) {
return product(
    positions, glScalarMergeMap<GLVector, double>(pr_map_arr)
) << sc_gen;
}
```

FIG. 15

290

13a

```
GLVectorGen pr_gen = getByInstrumentPortfolioSimulationGen(
    getLogNormalScenarioGen(iface), pr_map_arr
  );
```

292

B

```
GLMatrix mtf(num_scen, num_instr);
for (i=0; i<num_scen; ++i) {
  for (j=0; i<num_instr; ++j) {
    mtf[i,j] = (*pr_gen)[j];
  }
  ++pr_gen;
}
```

FIG. 16

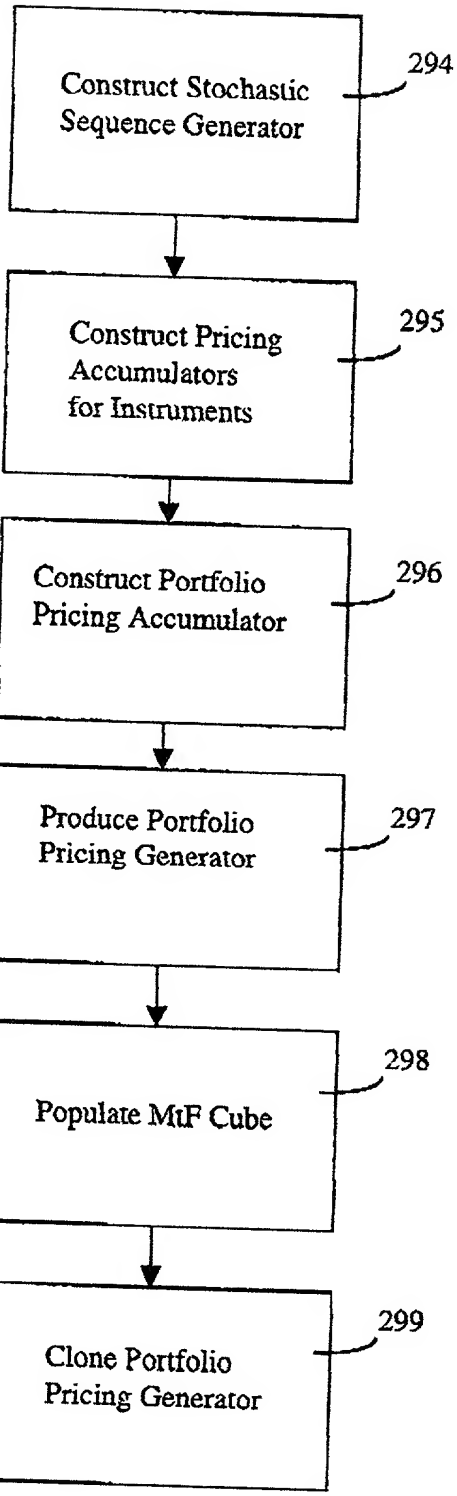
Construct Stochastic Sequence Generator
294
Construct Pricing Accumulators for Instruments
295
Construct Portfolio Pricing Accumulator
296
Produce Portfolio Pricing Generator
297
Populate MtF Cube
298
Clone Portfolio Pricing Generator
299

FIG. 17

- We have a set of instruments $I_1, I_2, \ldots, I_n$ with pricing accumulators $PrAcc_1, PrAcc_2, \ldots, PrAcc_n$. Each pricing accumulator depends on a number of risk factors and values of underlying instruments.

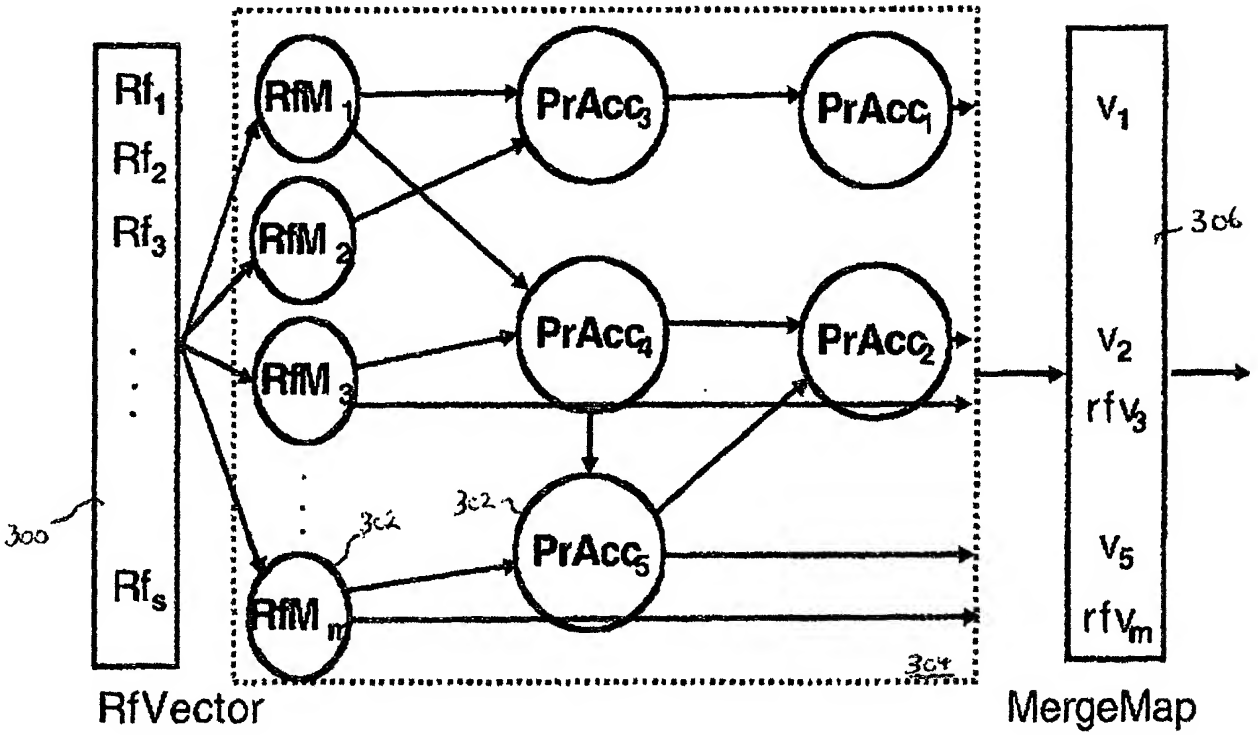

FIG. 18